Config Suite Documentation

Release 0.6.6

Equinor ASA & TNO

Jan 11, 2021

Contents:

Config Suite is the result of recognizing the complexity of software configuration.

CHAPTER 1

The User Guide

A guide through the universe of *Config Suite*.

1.1 Introduction

1.1.1 Philosophy

Config Suite is the result of recognizing the complexity of software configuration, both from a user and developer perspective. And our main goal is to be transparent about this complexity. In particular we aim at providing the user with confirmation when a valid configuration is given, concrete assistance when the configuration is not valid and up-to-date documentation to assist in this work. For a developer we aim at providing a suite that will handle configuration validity with multiple sources of data in a seamless manner, completely remove the burden of special casing and validity checking and automatically generate documentation that is up to date. We also believe that dealing with the complexity of formally verifying a configuration early in development leads to a better design of your configuration.

1.1.2 Features

- Validate configurations.
- Provide an extensive list of errors when applicable.
- Output a single immutable configuration object where all values are provided.
- Support for multiple data sources, yielding the possibility of default values as well as user and workspace configurations on top of the current configuration.
- Generating documentation that adheres to the technical requirements.
- No exceptions are thrown based on user-input (this is currently not true for validatiors).
- Context based validators and transformations.

1.1.3 Installation

The simplest way to fetch the newest version of *Config Suite* is via PyPI:

```
>>> pip install configsuite
```

1.1.4 License

1.2 Getting started

1.2.1 A first glance

For now we will just assume that we have a schema that describes the expected input. Informally say that we have a very simple configuration where one can specify ones name and hobby, i.e:

```
name: Espen Askeladd
hobby: collect stuff
```

You can then instantiate a suite as follows:

```
# ... configuration is loaded into 'input_config' ...

import configsuite

suite = configsuite.ConfigSuite(input_config, schema)
```

You can now check whether data provided in `input_config` is valid by accessing `suite.valid`.

```
if suite.valid:
    print("Congratulations! The config is valid.")
else:
    print("Sorry, the configuration is invalid.")
```

Now, given that the configuration is indeed valid you would probably like to access the data. This can be done via the `ConfigSuite` member named `snapshot`. Hence, we could change our example above to:

```
if suite.valid:
    msg = "Congratulations {name}! The config is valid. Go {hobby}."
    msg = msg.format(
        name=suite.snapshot.name,
        hobby=suite.snapshot.hobby,
    )
    print(msg)
else:
    print("Sorry, the configuration is invalid.")
```

And if feed the example configuration above the output would be

```
Congratulations Espen Askeladd! The config is valid. Go collect stuff.
```

However, if we changed the value of `name` to `13` (or even worse `["My", "name", "is kind", "of odd"]`) we would expect the configuration to be invalid and hence that the output would be `Sorry, the configuration is invalid`. And as useful as this is it would be even better to gain more detailed information about the errors.

```
>>> print(invalid_suite.errors)
(InvalidTypeError(msg=Is x a string is false on input '13', key_path=('hobby',),␣
↪layer=None),)
```

```
if suite.valid:
    msg = "Congratulations {name}! The config is valid. Go {hobby}."
    msg = msg.format(
        name=suite.snapshot.name,
        hobby=suite.snapshot.hobby,
    )
    print(msg)

else:
    print("Sorry, the configuration is invalid.")
    print(suite.errors)
```

```
Congratulations Espen Askeladd! The config is valid. Go collect stuff.
```

A first schema

The below schema is indeed the one used in our example above. It consists of a single collection containing the two keys `name` and `hobby`, both of which value should be a string.

```
from configsuite import types
from configsuite import MetaKeys as MK

schema = {
    MK.Type: types.NamedDict,
    MK.Content: {
        "name": {MK.Type: types.String},
        "hobby": {MK.Type: types.String},
    }
}
```

Notice the usage of the meta key `Type` to specify the type of a specific element and the usage of `Content` to specify the content of a container.

1.2.2 Types

In *Config Suite* we differentiate between *basic types* and *collections*. Basic types are single valued entities, while collections are data structures that can hold multiple basic types. In our first example the entire configuration was considered a collection (of type `Named dict`), while `name` and `hobby` are basic types. And while you can define arbitrary *basic types*, one cannot create new *collections* while using *Config Suite*.

Basic types

We will now give a brief introductory to the *basic types*. All of them can be utilized in a schema by utilizing the `MK.Type` keyword as displayed above. For an introduction to how one can implement user defined *basic types* we refer the reader to the advanced section.

String

We have already seen the usage of the `String` type above. It basically accepts everything considered a string in Python (defined by `six.string_types`).

Integer

An `Integer` is as the name suggests an integer.

Number

When a `Number` is specified any integer or floating point value is accepted.

Bool

Both boolean values `True` and `False` are accepted.

Date

A date in specified in ISO-format, `[YYYY]-[MM]-[DD]` that is.

DateTime

A date and time is expected in ISO-format (`[YYYY]-[MM]-[DD]T[hh]:[mm]:[ss]`).

Collections

We will now explore the supported *collections*. These will form the backbone of your configuration. In short, if you have a dictionary where you know the keys up front, you are looking for a *Named dict*, if you have dictionary with arbitrary keys, you are looking for a *Dict*. If you have a sequence of elements, you should check out *List*.

Named dict

We have already seen the usage of a *Named dict*. In particular, it allows for mapping values (of potentially different types) to names that we know up front. This allows us to represent them as attributes of the snapshot (or an sub element of the snapshot). In general, if you know the values of all of the keys up front, then a named dict is the right container for you.

```
from configsuite import types
from configsuite import MetaKeys as MK

schema = {
    MK.Type: types.NamedDict,
    MK.Content: {
        "owner": {
            MK.Type: types.NamedDict,
            MK.Content: {
                "name": {MK.Type: types.String},
                "credit": {MK.Type: types.Number},
                "insured": {MK.Type: types.Bool},
            },
        },
        "car": {
            MK.Type: types.NamedDict,
            MK.Content: {
                "brand": {MK.Type: types.String},
                "first_registered": {MK.Type: types.Date}
            },
        },
    },
}
```

the above example describes a configuration describing both an `owner` and a `car`. for the `owner` the `name`, `credit` and whether she is `insured` is to be specified, while for the `car` the `brand` and date it was `first_registered` is specified. a valid configuration could look something like this:

```
owner:
  name: Donald Duck
  credit: -1000
  insured: true

car:
  brand: Belchfire Runabout
  first_registered: 1938-07-01
```

and now, we could validate and access the data as follows:

```
# ... configuration is loaded into 'input_config' ...

import configsuite

suite = configsuite.ConfigSuite(input_config, schema)

if suite.valid:
    print("name of owner is {}".format(
        suite.snapshot.owner.name
    ))
    print("car was first registered {}".format(
```

(continues on next page)

(continued from previous page)

```
        suite.snapshot.car.first_registered
    ))
```

```
name of owner is Donald Duck
car was first registered 1938-07-01
```

Notice that since keys in a named dict are made attributes in the snapshot, they all have to be valid Python variable names.

List

Another supported container is the `List`. The data should be bundled together either in a Python `list` or a `tuple`. A very concrete difference of a Config Suite list and a Python list is that in Config Suite all elements are expected to be of the same type. This makes for an easier format for the user as well as the programmer when one is dealing with configurations. A very simple example representing a list of integers would be as follows:

```
import configsuite
from configsuite import types
from configsuite import MetaKeys as MK

schema = {
    MK.Type: types.List,
    MK.Content: {
        MK.Item: {
            MK.Type: types.Integer,
        },
    },
}

config = [1, 1, 2, 3, 5, 7, 13]

suite = configsuite.ConfigSuite(config, schema)

if suite.valid:
    for idx, value in enumerate(suite.snapshot):
        print("config[{}] is {}".format(idx, value))
```

```
config[0] is 1
config[1] is 1
config[2] is 2
config[3] is 3
config[4] is 5
config[5] is 7
config[6] is 13
```

A more complex example can be made by considering our example from the `NamedDict` section and imagining that an `owner` could have multiple `cars` that was to be contained in a list.

```
import datetime

import configsuite
from configsuite import types
from configsuite import MetaKeys as MK
```

(continues on next page)

(continued from previous page)

```
schema = {
    MK.Type: types.NamedDict,
    MK.Content: {
        "owner": {
            MK.Type: types.NamedDict,
            MK.Content: {
                "name": {MK.Type: types.String},
                "credit": {MK.Type: types.Number},
                "insured": {MK.Type: types.Bool},
            },
        },
        "cars": {
            MK.Type: types.List,
            MK.Content: {
                MK.Item: {
                    MK.Type: types.NamedDict,
                    MK.Content: {
                        "brand": {MK.Type: types.String},
                        "first_registered": {MK.Type: types.Date}
                    },
                },
            },
        },
    },
}

config = {
    "owner": {
      "name": "Donald Duck",
      "credit": -1000,
      "insured": True,
    },
    "cars": [
        {
          "brand": "Belchfire Runabout",
          "first_registered": datetime.date(1938, 7, 1),
        },
        {
          "brand": "Duckworth",
          "first_registered": datetime.date(1987, 9, 18),
        },
    ]
}

suite = configsuite.ConfigSuite(config, schema)

if suite.valid:
    print("name of owner is {}".format(suite.snapshot.owner.name))
    for car in suite.snapshot.cars:
        print("- {}".format(car.brand))
```

```
name of owner is Donald Duck
- Belchfire Runabout
- Duckworth
```

Notice that `suite.snapshot.cars` is returned as a `tuple`-like structure. It is iterable, indexable (`suite.snapshot.cars[0]`) and immutable.

Dict

The last of the data structures is the `Dict`. Contrary to the `NamedDict` one does not need to know the keys upfront and in addition the keys can be of other types than just `strings`. However, the restriction is that all the keys needs to be of the same type and all the values needs to be of the same type. The rationale for this is similar to that one of the list. Uniform types for arbitrary sized configurations are easier and better, both for the user and the programmer. A simple example mapping animals to frequencies are displayed below.

```
import configsuite
from configsuite import types
from configsuite import MetaKeys as MK

schema = {
    MK.Type: types.Dict,
    MK.Content: {
        MK.Key: {MK.Type: types.String},
        MK.Value: {MK.Type: types.Integer},
    },
}

config = {
    "donkey": 16,
    "horse": 28,
    "monkey": 13,
}

suite = configsuite.ConfigSuite(config, schema)
assert suite.valid

for animal, frequency in suite.snapshot:
    print("{} was observed {} times".format(animal, frequency))
```

```
donkey was observed 16 times
horse was observed 28 times
monkey was observed 13 times
```

As you can see, the elements of a `Dict` is accessible in `(key, value)` pairs in the same manner `dict.items()` would provide for a Python dictionary. The reason for not supporting indexing by key is `Dict`, contrary to `NamedDict`, is for dictionaries with an unknown set of keys. Hence, processing them as key-value-pairs is the only rational thing to do.

1.2.3 Configuration readiness

A very central concept in *Config Suite* is that of configuration readiness. Given that our configuration is indeed valid we can trust that `suite.snapshot` will describe all values as defined in the schema and that all the values are valid. Hence, we do not need to check for availability nor correctness of the configuration.

Readable

The concept of configuration readiness implies if one specified a value in the schema, one is to expect that that piece of data is indeed present in the snapshot. But what if the configuration fed to the suite is not valid? If the errors appear in basic types, one can still access all the data as expected (i.e. `config.snapshot.owner.name` from the car example above). However, if a container is of the wrong type one cannot guarantee such a thing. In particular, if we bring back the single-car example from above and consider the following configuration:

```
owner:
  name: Donald Duck
  credit: -1000
  insured: true

car:
  - my first car
  - my second car
```

The `car` data is completely off and there is no way one could provide a reasonable value for `config.snapshor.car.brand`. In such scenarios the configuration is deemed *unreadable*. There is a special marker for this, namely `ConfigSuite.readable`. If `readable` is true, then the snapshot can be built and all the entire configuration can be accessed. However, if the suite is not `readable` and one tries to fetch the snapshot an `AssertionError` will be raised.

Note that all valid suites also are readable. And that all unreadable suites also are invalid.

1.2.4 Allow None

For certain configurations it may be reasonable to provide `None` as the value. The `AllowNone` type is only valid for `BasicType`'s, not for containers. Setting `AllowNone` for anything but `BasicType` will result in an invalid schema.

Let us see how the owner section of the `cars` schema could be configured in order for a configuration with `None` to pass.

```
import configsuite
from configsuite import types
from configsuite import MetaKeys as MK

schema = {
    MK.Type: types.NamedDict,
    MK.Content: {
        "owner": {
            MK.Type: types.NamedDict,
            MK.Content: {
                "name": {MK.Type: types.String},
                "credit": {
                    MK.Type: types.Number,
                    MK.AllowNone: True,
                    MK.Required: False,
                },
                "insured": {MK.Type: types.Bool},
            },
        },
    },
}

config = {
    "owner": {
            "name": "Scrooge",
            "credit": None,
            "insured": False,
    },
}
```

(continues on next page)

(continued from previous page)

```
suite = configsuite.ConfigSuite(config, schema)
assert suite.valid

owner = suite.snapshot.owner
print("{} has a credit of {}".format(owner.name, owner.credit))
```

```
Scrooge has a credit of None
```

1.2.5 Allow Empty

Occasionally one would like to require variable length containers (lists and dicts) to have at least one element. This can be implemented with a validator on the container (see *Validators*). However, as this feature have been requested multiple times, we've decided to implement explicit support for it without having to implement your own validator. Note that by default all containers are allowed to be empty.

```
import configsuite
from configsuite import types
from configsuite import MetaKeys as MK

schema = {
    MK.Type: types.List,
    MK.AllowEmpty: False,
    MK.Content: {
        MK.Item: {MK.Type: types.Integer},
    },
}

non_empty_suite = configsuite.ConfigSuite([0, 1, 2, 3, 4], schema)
assert non_empty_suite.valid

empty_suite = configsuite.ConfigSuite([], schema)
assert not empty_suite.valid
```

1.2.6 Default values

So far all entries in your configuration file have been mandatory to fill in. And if some key in a Named dict would be missing a `MissingKeyError` would be registered. However, this is not always the wanted behaviour. By using the `MetaKeys.Required` option you can control whether a key is indeed required. You could change the `cars` schema above such that `credit` would be optional as follows:

```
from configsuite import types
from configsuite import MetaKeys as MK

schema = {
    MK.Type: types.NamedDict,
    MK.Content: {
        "owner": {
            MK.Type: types.NamedDict,
            MK.Content: {
                "name": {MK.Type: types.String},
                "credit": {
                    MK.Type: types.Number,
```

(continues on next page)

(continued from previous page)

```
                    MK.Required: False,
                },
                "insured": {MK.Type: types.Bool},
            },
        },
        "cars": {
            MK.Type: types.List,
            MK.Content: {
                MK.Item: {
                    MK.Type: types.NamedDict,
                    MK.Content: {
                        "brand": {MK.Type: types.String},
                        "first_registered": {MK.Type: types.Date}
                    },
                },
            },
        },
    },
}
```

And then if no `credit` was specified a `MissingKeyError` would not be registered. However, recall the principle of configuration readiness. Since, the programmer should not have to special case whether or not the value is present in the `snapshot`. The `snapshot` is always built based on the schema and hence `suite.snapshot.owner.credit` would indeed be an attribute independently of whether the user has configured it. In this scenario the value of `suite.snapshot.owner.credit` would be `None`.

`ConfigSuite` requires that any entity that is not required must also allow `None`.

How to specify default values

There exists two ways of providing default values in Config Suite. You are to either specify it in the schema via the keyword `MetaKeys.Default`. This has the advantage of being able to provide default values for `BasicTypes` within containers. The disadvantage is that you would need to edit the code to change the default values and hence site or project specific defaults are not suited for this purpose. The second way of specifying default are via `layers`.

Note that no element should be both required and have a given `Default` value.

Schema defaults

The default value for any `BasicType` may be set within the schema itself by using the `MK.Default` key. The `Type` of `MK.Default` must be consistent with the schema configuration, otherwise it will not be valid. Please note that any *Validators* or *Transformations* are applied to default values as well, and they can be regarded as if they were coming directly from the user. Setting a default value implies that it is not required, and thus `MK.Required` must be set to `False`.

Let us see how the `owner` section could be configured with default value for the `credit`:

```
import configsuite
from configsuite import types
from configsuite import MetaKeys as MK

schema = {
    MK.Type: types.NamedDict,
    MK.Content: {
```

(continues on next page)

(continued from previous page)

```
        "owner": {
            MK.Type: types.NamedDict,
            MK.Content: {
                "name": {MK.Type: types.String},
                "credit": {
                    MK.Type: types.Number,
                    MK.Default: 0,
                    MK.Required: False,
                    MK.AllowNone: True,
                },
                "insured": {MK.Type: types.Bool},
            },
        },
    },
}

config = {
    "owner": {
            "name": "Scrooge",
            "insured": False,
    },
}

suite = configsuite.ConfigSuite(config, schema)
assert suite.valid

owner = suite.snapshot.owner
print("{} has a credit of {}".format(owner.name, owner.credit))
```

```
Scrooge has a credit of 0
```

Layers

Layers is a fundamental concept in *Config Suite* that enables you to retrieve configurations from multiple sources in a consistent manner. It can be utilized to give priority to different sources, being application defaults, installation defaults, project or user settings, as well as case specific configuration. It can also be utilized to represent changes in configuration from a UI in a consistent manner.

In short, a layer is, a possibly incomplete, configuration source. Multiple layers can be stacked on top of each other to form a single configuration. In such a stack, top layers take precedence over lower layers. For each of the types there are specific rules for how that type is merged when multiple layers are combined into a single value.

Layers can be passed to a suite via the keyword argument `layers`. In particular, if constructed as follows

```
import configsuite
from configsuite import MetaKeys as MK
from configsuite import types

schema = {
    MK.Type: types.NamedDict,
    MK.Content: {
        "a": {MK.Type: types.Integer},
        "b": {MK.Type: types.Integer},
        "c": {MK.Type: types.Integer},
    }
```

(continues on next page)

(continued from previous page)

```
}

config = { "a": 0 }
middle_layer = { "a": 1, "b": 1 }
bottom_layer = { "a": 2, "b": 2, "c": 2}

suite = configsuite.ConfigSuite(
    config,
    schema,
    layers=(bottom_layer, middle_layer),
)

print(suite.valid)
print(suite.snapshot.a)
print(suite.snapshot.b)
print(suite.snapshot.c)
```

```
True
0
1
2
```

This will result in the layers `(bottom_layer, middle_layer, config)`, where elements in `config` takes precedence over the two other layers and the elements in `middle_layer` over elements in `bottom_layer`.

Basic types

Basic types are simply overwritten and only the value from the top most layer specifying that value is kept.

Named dicts and dicts

Named dicts and dicts are by default joined in an update kind of fashion. All the values are joined recursively, key by key. This implies that for the `cars`-example with the following layers:

```
# Lower level
owner:
  name: Donald Duck
  credit: 100
```

```
# Upper level
owner:
  name: Scrooge McDuck
  insured: True
```

would result in the following after being merged:

```
# Merged configuration
owner:
  name: Scrooge McDuck
  credit: 100
  insured: True
```

Lists

Lists are by default appended, with the top layer elements appearing after lower levels. If we again lock at the `cars`-example:

```
# Lower level
cars:
  -
    brand: Belchfire Runabout
    first_registered: 1938-7-1
  -
    brand: Duckworth
    first_registered: 1987-9-18
```

```
# Upper level
cars:
  -
    brand: Troll
    first_registered: 1956-11-6
```

would result in the following after being merged:

```
# Merged configuration
cars:
  -
    brand: Belchfire Runabout
    first_registered: 1938-7-1
  -
    brand: Duckworth
    first_registered: 1987-9-18
  -
    brand: Troll
    first_registered: 1956-11-6
```

1.2.7 Documentation generation

The available functionality for generating documentation is still rather limited, but will continously be improved upon.

Programatically

You can pass your schema to `configsuite.docs.generate` and it will generate documentation as reStructuredText.

Sphinx

Config Suite includes a sphinx extension for generating documentation. It has only been tested to work with the default sphinx theme; `Alabaster`. The extension must be included in the `conf.py` file:

```
extensions += "configsuite.extension.ext"
```

The sphinx directive `configsuite` can then be added in your documentation as follows:

```
.. configsuite::
    :module: module.function
```

Where the `function` value points to a function that returns a valid *Config Suite* schema.

The generated documentation will contain a section for each container and can be expanded by clicking the header. There is fairly limited graphical designs added by default, but you are free to include more. The generated html files will consist of four `classes`; `cs_top_container`, `cs_container`, `cs_header`, `cs_content` and `cs_children`. The `cs_top_container` contains the entire configuration, while each subsection in the schema will have at least one of each of the remaining classes.

The `cs_container` is the main container holding one of each of `cs_header` and `cs_content`. The `cs_content` contains the text from `MK.Description` and includes the messages from any `Validators` and `Tansformations`. The `cs_children` class furthermore contains a new node for each sub-element that is built with the same classes above.

Given the nature of the schema's one would typically end up with many `cs_containers`. We have tried to facilitate specific customization if the user would like to have that, by including the possibility of unique `id`'s for every container. The id is for e.g. `cs_{}_container`, where `{}` will be replaced by the title.

You can then include a customized `.css` file that acts on each item.

1.2.8 Validators

Validators enables validation beyond the type validation. As a first example, let us say that you have a value in your configuration that should only contain characters from the alphabet and spaces. This would be a quite natural validation of the `name` field in our first example.

First, you need to write a function that validates the requirements above and returns its result as a boolean.

```
import configsuite


@configsuite.validator_msg("Is x a valid name")
def _is_name(name):
    return all(char.isalpha() or char.isspace() for char in name)
```

Notice the decorator `validator_msg`. This adds a statement regarding the purpose of the validator to the validator and a statement regarding the result of the validation to the returned result. These messages are used both if a validator fails to register errors as well as to generate documentation. In particular:

```
print(_is_name.msg)
print(_is_name("1234").msg)
print(_is_name("My Name").msg)
```

```
Is x a valid name
Is x a valid name is false on input '1234'
Is x a valid name is true on input 'My Name'
```

Afterwards, you can add this to your schema as follows:

```
from configsuite import types
from configsuite import MetaKeys as MK

schema = {
    MK.Type: types.NamedDict,
```

(continues on next page)

(continued from previous page)

```
    MK.Content: {
        "name": {
            MK.Type: types.String,
            MK.ElementValidators: (_is_name,),
        },
        "hobby": {MK.Type: types.String},
    }
}
```

Notice that we do not have to check that the input is a string. This is because type validation is always carried out first and the validator is only applied if the type validation succeeded.

1.2.9 Transformations

Transformations enables changing the data in the merged configuration before it is validated and the snapshot becomes accessible. A simple example of this is that you would like to support scientific notation for numbers in your configuration files. This is a well-known short coming of *PyYAML*. In particular, you would like the `cars` example to support the following:

```
owner:
  name: Donald Duck
  credit: 1e10
  insured: true
cars: []
```

However, loading the above from a `yml`-file would yield the following data:

```
config = {
    "owner": {
      "name": "Donald Duck",
      "credit": "1e10",
      "insured": True,
    },
    "cars": [],
}
```

Note that the value of `credit` is a string. However, it is easy to write a transformer for this purpose.

```
import configsuite


_num_convert_msg = "Tries to convert input to a float"
@configsuite.transformation_msg(_num_convert_msg)
def _to_float(num):
    return float(num)
```

And now, we can insert this into the schema as follows:

```
from configsuite import types
from configsuite import MetaKeys as MK

schema = {
    MK.Type: types.NamedDict,
    MK.Content: {
        "owner": {
```

(continues on next page)

(continued from previous page)

```
            MK.Type: types.NamedDict,
            MK.Content: {
                "name": {MK.Type: types.String},
                "credit": {
                    MK.Type: types.Number,
                    MK.Required: False,
                    MK.AllowNone: True,
                    MK.Transformation: _to_float,
                },
                "insured": {MK.Type: types.Bool},
            },
        },
        "cars": {
            MK.Type: types.List,
            MK.Content: {
                MK.Item: {
                    MK.Type: types.NamedDict,
                    MK.Content: {
                        "brand": {MK.Type: types.String},
                        "first_registered": {MK.Type: types.Date}
                    },
                },
            },
        },
    },
}
```

Last, we observe that

```
suite = configsuite.ConfigSuite(
    config,
    schema,
)

print(suite.valid)
print(suite.snapshot.owner.credit)
```

```
True
10000000000.0
```

As a final note about transformations it should be said that currently *Config Suite* does not validate readability in between transformations. This implies that if a transformations has the capability of changing the data type of a collection, then the promise of the transformations being provided with data of the correct type is only true as long as the transformations preserve this while being applied.

1.3 Advanced Usage

1.3.1 Creating your own types

Config Suite supports you creating your own basic types. This is done by creating a new instance of `configsuite.BasicType`. The constructor takes a type name, as well as a validator of the type. For instance, the `Date` type in Config Suite could be implemented as follows:

```
import configsuite
import datetime

@configsuite.validator_msg("Is x a date")
def _is_date(x):
    return isinstance(x, datetime.date)

Date = configsuite.BasicType("date", _is_date)
```

After this, you can use `Date` as a `MetaKeys.Type` value in your schema as displayed in the `cars`-schema.

```
import yaml
from configsuite import MetaKeys as MK

schema = {MK.Type: Date}
config = yaml.load("1988-06-05")

suite = configsuite.ConfigSuite(config, schema)

print(suite.valid)
print(suite.snapshot)
```

```
True
1988-06-05
```

1.3.2 Context validators

Context validators enables validation of an entry based on the value of entries located elsewhere in the configuration. This is a two-step procedure; where the first step is a consumer-defined function for extracting context of a snapshot. The given snapshot is guaranteed to be readable and is extracted after all transformations have been applied. Then, after an element have been validated recursively and the element validator has been applied, if the element is still deemed valid, the context validator is applied.

The typical application of a context validator is when you have multiple, central concepts in your configuration. An example would be a configuration of students and classes. You want to enable specification of both students and classes, yet you also want to list the students attending each class.

```
import collections

import configsuite
from configsuite import MetaKeys as MK
from configsuite import types


_student_schema = {
    MK.Type: types.List,
    MK.Content: {
        MK.Item: {
            MK.Type: types.NamedDict,
            MK.Content: {
                "name": {MK.Type: types.String},
                "age": {MK.Type: types.Integer},
                "favourite_lunch": {MK.Type: types.String}
            },
        },
```

(continues on next page)

(continued from previous page)

```
    },
}


@configsuite.validator_msg("Is x a student name")
def _is_student(name, context):
    return name in context.student_names


_course_schema = {
    MK.Type: types.List,
    MK.Content: {
        MK.Item: {
            MK.Type: types.NamedDict,
            MK.Content: {
                "name": {MK.Type: types.String},
                "max_size": {MK.Type: types.Integer},
                "students": {
                    MK.Type: types.List,
                    MK.Content: {
                        MK.Item: {
                            MK.Type: types.String,
                            MK.ContextValidators: (_is_student,),
                        },
                    },
                },
            },
        },
    },
}


schema = {
    MK.Type: types.NamedDict,
    MK.Content: {
        "students": _student_schema,
        "courses": _course_schema,
    },
}


def _extract_student_names(snapshot):
    Context = collections.namedtuple("Context", ("student_names",))
    student_names = tuple(
        student.name for student in snapshot.students
    )
    return Context(student_names=student_names)
```

Note that we have split the schema in two in this example to keep it a bit more manageable. Also, observe how `_is_student` takes both a `name` (which we know is a `string`) as well as a `context`. Besides that, we had to implement a function that extracts the `context` from a snapshot. Notice that we are in complete control of the `context`, which means that we could have returned the entire snapshot, or a tuple containing just the `student` names. The first is not recommended as it is good to be conscious regarding what the context contains. In particular, a user will have to carry the same amount of information in their head while auditing a configuration file. The latter, because we recommend to make the context extendable without having to change all the context validators. And by putting all student names under an attribute, we achieve just that.

Now, to create a suite with the configuration `config`, we do as follows:

```
config = {
    "students": [
        {
            "name": "Per",
            "age": 21,
            "favourite_lunch": "graut",
        },
        {
            "name": "Espen",
            "age": 17,
            "favourite_lunch": "troll",
        },
    ],
    "courses": [
        {
            "name": "adventures-101",
            "max_size": 50,
            "students": ["Per", "Espen"],
        },
    ],
}

suite = configsuite.ConfigSuite(
    config,
    schema,
    extract_validation_context=_extract_student_names,
)

print(suite.valid)
```

```
True
```

However, if we add a course with an unknown student:

```
invalid_suite = suite.push({
    "courses": [
        {
            "name": "impossible-101",
            "max_size": 0,
            "students": "Pål",
        },
    ],
})

print(invalid_suite.valid)
print(invalid_suite.errors)
```

```
False
(InvalidTypeError(msg=Is x a list is false on input 'Pål', key_path=('courses', 0,
↪'students'), layer=1),)
```

1.3.3 Context transformations

Context transformations allows for transforming elements based on the values of other elements. This was implemented to support the following scenario; you want to define variables in one part of the configuration and then substitute values in another part based on these variables. We will now display a simple implementation of such a system.

First, we must implement functionality for given a `template` and `definitions` to render the templates. That can be done as follows:

```
import collections
import copy
import jinja2

import configsuite
from configsuite import MetaKeys as MK
from configsuite import types


# To avoid collision with dict and set syntax in yaml
_VAR_START = "<"
_VAR_END = ">"


def _render_variables(variables, jinja_env):
    """Repeatedly render the variables to support the scenario when one
    variable refers to another one.
    """
    variables = copy.deepcopy(variables)
    for _ in enumerate(variables):
        rendered_values = []
        for key, value in variables.items():
            try:
                variables[key] = jinja_env.from_string(value).render(variables)
                rendered_values.append(variables[key])
            except TypeError:
                continue

    if any([_VAR_START in val for val in rendered_values]):
        raise ValueError("Circular dependencies")

    return variables


def _render(template, definitions):
    """Render a template with the given definitions."""
    if definitions is None:
        definitions = {}

    variables = copy.deepcopy(definitions)
    jinja_env = jinja2.Environment(
        variable_start_string=_VAR_START, variable_end_string=_VAR_END,␣
↪autoescape=True
    )

    try:
        variables = _render_variables(variables, jinja_env)
        jinja_template = jinja_env.from_string(template)
```

(continues on next page)

(continued from previous page)

```
        return jinja_template.render(variables)
    except TypeError:
        return template


@configsuite.transformation_msg("Renders Jinja template using definitions")
def _context_render(elem, context):
    return _render(elem, definitions=context.definitions)
```

Second, we must implement a context extractor.

```
def extract_templating_context(configuration):
    Context = collections.namedtuple("TemplatingContext", ["definitions"])
    definitions = {key: value for (key, value) in configuration.definitions}
    return Context(definitions=definitions)
```

Third, we define the schema.

```
schema = {
    MK.Type: types.NamedDict,
    MK.Content: {
        "definitions": {
            MK.Type: types.Dict,
            MK.Content: {
                MK.Key: {MK.Type: types.String},
                MK.Value: {MK.Type: types.String},
            },
        },
        "templates": {
            MK.Type: types.List,
            MK.Content: {
                MK.Item: {
                    MK.Type: types.String,
                    MK.ContextTransformation: _context_render,
                }
            },
        },
    },
}
```

And then, given the following yaml-configuration:

```
definitions:
  animal: pig
  habitants: <animal>, cow and monkey
  color: blue
  secret_number: "42"
templates:
  - This is a story about a <animal>.
  - It had a <color> house.
  - And the password to enter was <secret_number>.
  - If you entered the house you would meet: <habitants>.
  - The end.
```

we obtain the following rendered `templates` after feeding the `config`, `schema` and the `extract_templating_context` through `configsuite.ConfigSuite`.

```
import yaml

config = yaml.safe_load("""
definitions:
  animal: pig
  habitants: <animal>, cow and monkey
  color: blue
  secret_number: "42"
templates:
  - This is a story about a <animal>.
  - It had a <color> house.
  - And the password to enter was <secret_number>.
  - "If you entered the house you would meet: <habitants>."
  - The end.
""")
suite = configsuite.ConfigSuite(
    config,
    schema,
    extract_transformation_context=extract_templating_context,
)

print(suite.valid)
```

```
True
```

Furthermore, if we print the rendered templates we get the following:

```
for line in suite.snapshot.templates:
    print(line)
```

```
This is a story about a pig.
It had a blue house.
And the password to enter was 42.
If you entered the house you would meet: pig, cow and monkey.
The end.
```

Notice that we can now merge multiple layers, with definitions in higher levels taking precedence.

A note on contexts

In these section we cover some advance topics that should be used with care. The contexts due to the fact that parts of the configuration can no longer be validated independently. Which implies that the user might have to make changes far apart to keep a configuration file consistent, data that is naturally displayed together in a UI cannot be validated without taking the rest of the configuration into account and the difficulty of understanding your configuration increases.

1.3.4 Layer transformations

A layer transformation is applied to an element of a layer **before** the various layers are merged. The observant reader might notice that hence the layer transformations provide no real benefit over a standard transformation for Basic Types. The natural application of a layer transformation is to transform a collection such that merging can be carried out. Due to this, *Config Suite* can give no guarantee what so ever on the content of the data provided to a layer transformation.

A natural application of layer transformations is to support ranges where lists of integers are expected. In particular, one would like to be able to write `1-3, 5-7, 9` and get `[1, 2, 3, 5, 6, 7, 9]` as a result. Assume one

implements a function `_realize_list` that takes as input a string of ranges and singletons and returns a list as in the example above. And that one in addition decorates it with a `transformation_msg`. Then, the following schema

```
from configsuite import MetaKeys as MK
from configsuite import types

{
    MK.Type: types.List,
    MK.LayerTransformation: _realize_list,
    MK.Content: {MK.Item: {MK.Type: types.Integer}},
}
```

gives the specification of a list of integers for which one can instead provide strings of ranges and singletons in some of the layers. For a full implementation, where in addition the final list (after the layers have been merged) is sorted and duplicates are removed we refer the reader to the test data.

Note that for the layer transformations to give the intended functionality they are applied in a top down manner. This is another distinction from the other transformations (including the context transformations further down) that are all applied in a bottom up manner.

1.4 Deprecated behaviour

1.4.1 Specifying AllowNone and Default

In *Config Suite* 0.6 the schema keywords *AllowNone* and *Default* was introduced. *AllowNone* is used to indicate whether a basic element can take the value *None* and is defaulted to *False*. And *Default* can be used to provide default values for elements and it defaults to *None*. It should be noted that there is a strong correlation between *Default*, *AllowNone* and *Required*. In particular, the later can be deduced from the first two. In short, if an element is required it should neither allow *None* values nor have a default. Because if so, it would not be required.

Here is a complete table of valid combinations of the three options:

Table 1: Valid combinations

AllowNone	Default	Required
True	True	False
True	False	False
False	True	False
False	False	True

1.4.2 Required is deprecated

In the transition to *Config Suite* 0.6 specifying elements in the schema as *Required* is deprecated. In particular, it is superseded as described above by specifying *AllowNone* and *Default*. The plan is that 0.6 will still function with *Required* in the schemas to ease the process of introducing *AllowNone* and *Default*. However, upon initialization *ConfigSuite* now accepts an optional, boolean argument named *deduce_required*. If *deduce_required* is set to *False* (which is the default value) a deprecation warning will be raised. After the transition to using *AllowNone* and *Default* has been made (which the reader is heavily encouraged to finish before considering the deprecation of *Required*) the consumer is expected to toggle *deduce_required* to *True*. At this point a new deprecation warning will be raised upon creation of a *ConfigSuite* instance if the schema contains a *Required* specification. However, they are now all safe to remove and you are indeed encouraged to do so.

Then in the 0.7 release *deduce_required* will default to *None*. If *True* is passed instead we will raise a deprecation warning and for any other value we will raise an exception. The idea is that in 0.7 you can safely stop setting *deduce_required* and then be ready for 0.8, where *deduce_required* will not be recognized as an optional argument anymore.

CHAPTER 2

Additional examples

For additional examples we refer the reader to the test data.

CHAPTER 3

Release Notes

An overview of the *Config Suite* releases.

3.1 Release Notes

3.1.1 dev

3.1.2 0.6.6 (2021-01-05)

Miscellaneous

- Stop running CI on Travis and move PyPI deploy to GitHub Actions

3.1.3 0.6.5 (2020-11-27)

New features

- Add possibility in schema to specify whether a variable length container (list or dict) is allowed to be empty

Miscellaneous

- Replace usage of deprecated inspect methods

3.1.4 0.6.4 (2020-09-25)

Improvements

- Validate that content is specified in schema if and only if element is a container.

Deprecations

- Drop Python 2.7 support

Dependencies

- Remove dependency on six

3.1.5 0.6.3 (2020-07-01)

Improvements

- Have the Sphinx plugin only render examples if they are provided

Bugfix

- Make the Sphinx plugin's CSS changes local to the plugin

3.1.6 0.6.2 (2020-06-10)

Improvements

- Better validation message for incompatible required value

Bugfix

- Pass on *deduce_required* when pushing

3.1.7 0.6.1 (2020-06-03)

Improvements

- Add empty line between keys and values in auto generated rst documentation to avoid warning on empty descriptions

Bugfix

- Default all containers to the empty container

3.1.8 0.6.0 (2020-05-28)

New features

- Specify elements to be allowed to take None as value
- Specify default values for elements in the schema
- Generate documentation from schema a Sphinx plugin

Improvements

- Python 3.8 support

Deprecations

- Specifying elements as Required in the schema
- Python 3.4 support

Miscellaneous

- Use *flake8* as part of the CI pipeline
- Have the CI pipeline ensure that the docs builds without warnings

- Add GitHub actions as another CI provider
- Validate internal meta schema in the tests

Dependencies

- The following install dependencies have been added: *docutils*, *PyYAML* and *sphinx*

3.1.9 0.5.3 (2019-11-14)

Improvements

- Make sure all examples in the documentation is valid and run them in the test suite

3.1.10 0.5.2 (2019-08-30)

Improvements

- Various improvements to the documentation

3.1.11 0.5.1 (2019-06-14)

Improvements

- Fix typos in the documentation

3.1.12 0.5.0 (2019-06-13)

New features

- Support context validation for containers

Improvements

- Allow for chaining *BooleanResults*, and hence transformations and validators

3.1.13 0.4.2 (2019-05-19)

Miscellaneous

- Improve information in setup.py

3.1.14 0.4.1 (2019-05-19)

Bugfixes

- Extractors are now passed to new suite when pushing layers

New features

- Initial documentation written and hosted at read the docs

Miscellaneous

- Minor code improvements

3.1.15 0.4.0 (2019-05-07)

New features

- Support for context validation
- Support for layer transformations, transformations and context transformations

Improvements

- Remove layer validation
- Accept unicode strings as strings
- No sorting of dict keys

3.1.16 0.3.1 (2019-04-29)

Bugfixes

- Fix various errors regarding imports

3.1.17 0.3.0 (2019-04-26)

Bugfixes

- Fix docs import in configsuite's init-file

New features

- New basic types *Date* and *DateType*

Dependencies

- Add six to Python 2 dependencies

3.1.18 0.2.1 (2019-04-12)

Bugfixes

- Add description to meta schema

Miscellaneous

- Various code improvements due to PyLint

3.1.19 0.2.0 (2019-04-03)

New features

- Documentation generating capabilities from the specification
- Support for layered configurations

3.1.20 0.1.0 (2018-11-08)

New features

- Initial validation and snapshot implementation
- Validation of schema
- Support for basic types: int, string, number and bool
- Support for containers: list, named_dict and dict
- Support for non-required dict keys

CHAPTER 4

The API Documentation

In-deepth documentation of the API of *Config Suite*.

4.1 The main interface

class configsuite.**ConfigSuite**(*raw_config*, *schema*, *layers=()*, *extract_validation_context=<function ConfigSuite.<lambda>>*, *extract_transformation_context=<function ConfigSuite.<lambda>>*, *deduce_required=False*)

A *Suite* exposing the functionality of Config Suite in a unified manner.

The intended usage of Config Suite is via this immutable suite. It is constructed with a *schema* describing the structure of a configuration, together with a *raw_config* and possibly additional *layers*.

Parameters

- **raw_config** – The configuration taking precedence.
- **schema** – A description of the structure of a valid configuration, together with actions that are to be carried out.
- **layers** (*iterable of layers, optional*) – Additional layers of configuration. A layer takes precedence over all other layers following it in the given sequence. Note that *raw_config* takes precedence over all elements of *layers*.
- **extract_validation_context** (*callable, optional*) – Callable that extracts the context used for validation. The callable is given a snapshot of the configuration as argument. Defaults to the constant function always returning *None*.
- **extract_transformation_context** (*callable, optional*) – Callable that extracts the context used for transformations. The callable is given a snapshot of the configuration as argument. Defaults to the constant function always returning *None*.
- **deduce_required** (*bool, optional*) – Boolean that enables future behaviour of deducing whether a schema entry is *required* by inspecting *allow_none* and *default*. In

particular, using *required* in schemas as well as not setting *deduce_required=True* is deprecated.

Raises *TypeError, KeyError, ValueError* – Approperiate errors are raised if provided with an invalid schema. Note that errors are independent of *raw_config* and *layers* and hence not user input responsive.

errors
An iterable of *configsuite.errors* that occurred during validation. Is always empty if *valid* is *True* and non-empty otherwise.

push(*raw_config*)
Builds a new suite with *raw_config* on top of the current suite.

Builds a new suite with the same schema, but with *raw_config* on top of the layers in the current suite.

Parameters **raw_config** – A configuration that is to take precedence over all layers in the current suite.

Returns

Return type A new *ConfigSuite* with *raw_config* as the first layer.

readable
A boolean indicating whether the resulting configuration was deemed readable. Is always *True* if *valid* is *True*. For the consequences of being readable, see the documentation of *snapshot*.

snapshot
A complete, immutable representation of the resulting configuration.

Raises `AssertionError` – Raised on access if *suite* is not *readable*.

valid
A boolean indicating whether the resulting configuration was deemed valid according to the schema.

Python Module Index

C

Index